ORACLE

Compiler-Assisted Object Inlining with Value Fields

Rodrigo Bruno, INESC-ID / Técnico - ULisboa*

Vojin Jovanovic, Oracle Labs Switzerland

Christian Wimmer, Oracle Labs USA

Gustavo Alonso, Department of Computer Science, ETH Zurich

* work done while at Oracle Labs Switzerland and ETH Zurich

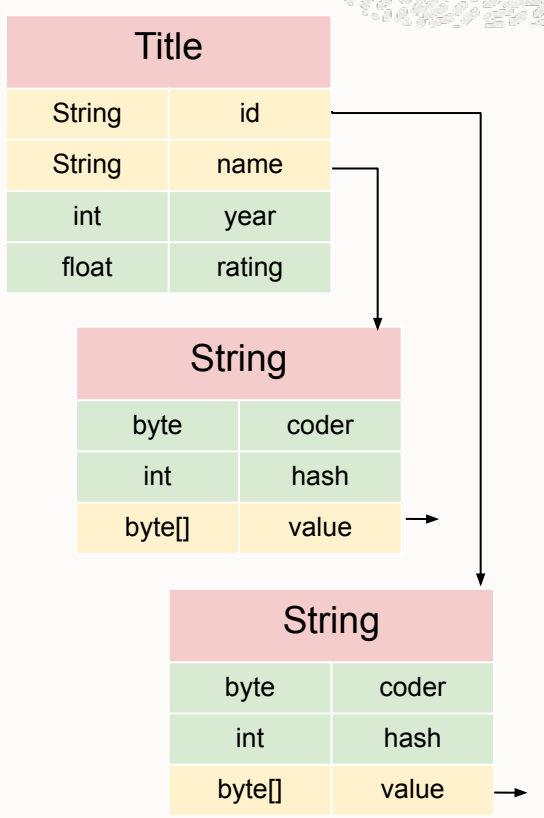
Data *objectification*

Title ID	Name	Year	Rating
...	...	...	...
tt0110912	Pulp Fiction	1994	8.9
...	...	...	...

Database
(1 entry)

Data *objectification*

Title ID	Name	Year	Rating
...	...	...	...
tt0110912	Pulp Fiction	1994	8.9
...	...	...	...



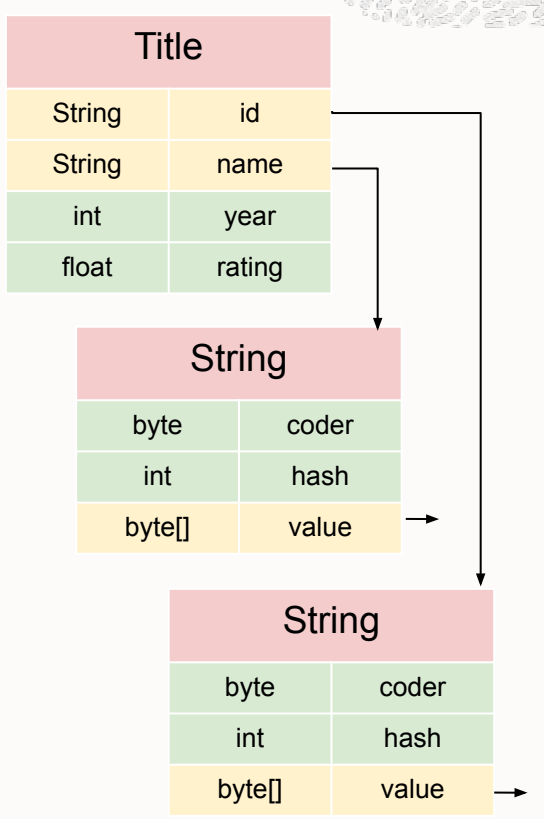
Database
(1 entry)

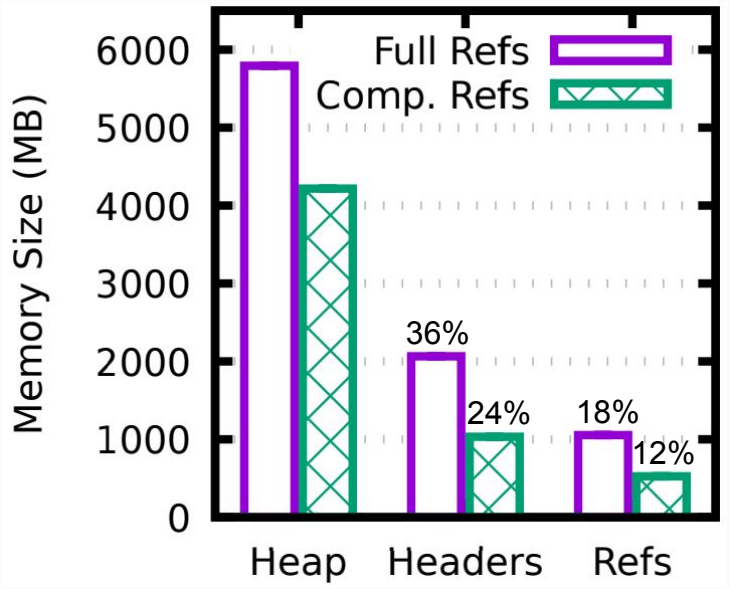
Java Heap
(5 objects, 4 references)

Data *objectification*

Data *objectification*

Title ID	Name	Year	Rating
...	...	...	...
tt0110912	Pulp Fiction	1994	8.9
...	...	...	...





‘unusable’ mem ~ 36-54%

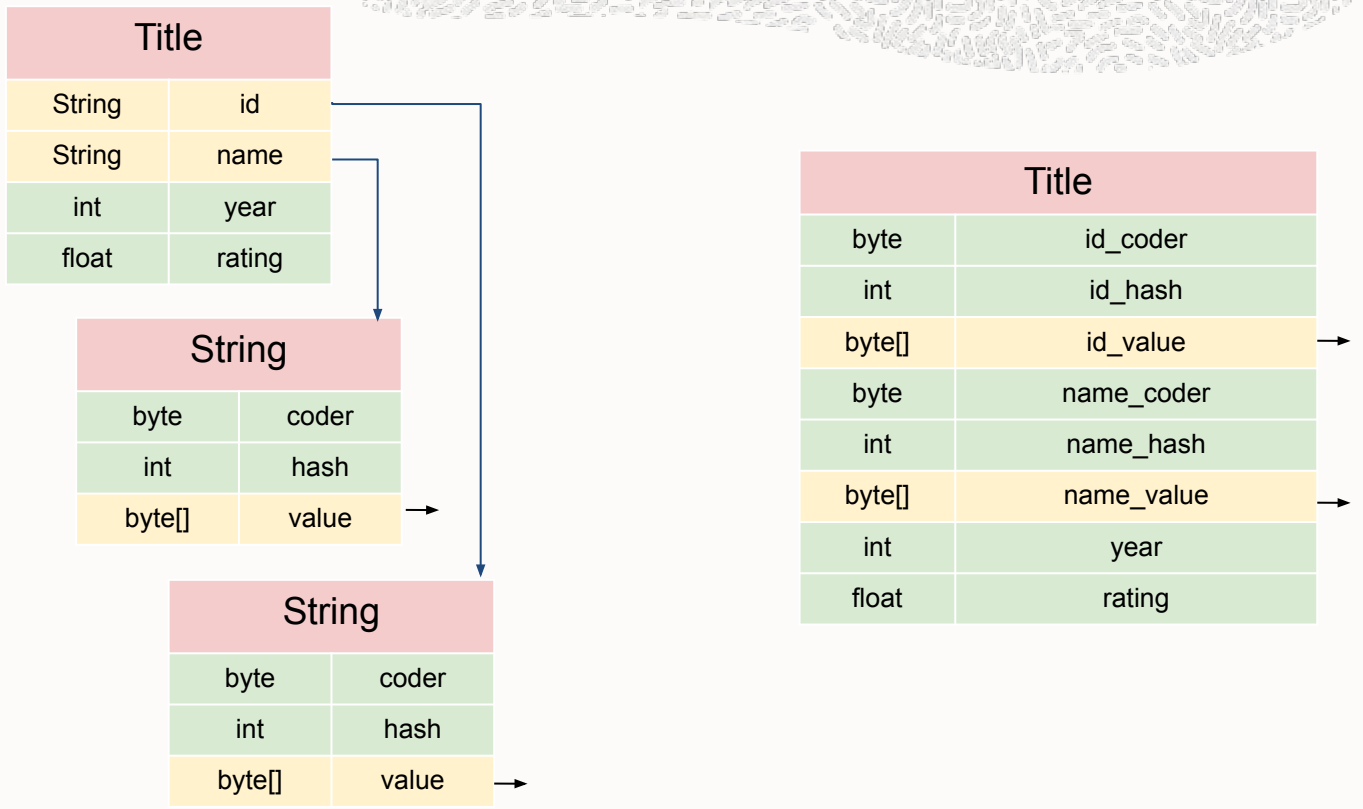
Database
(1 entry)

Java Heap
(5 objects, 4 references)

Data *objectification*

Data *objectification*

Title ID	Name	Year	Rating
...	...	...	...
tt0110912	Pulp Fiction	1994	8.9
...	...	...	...



Database
(1 entry)

Java Heap
(5 objects, 4 references)

Java Heap
(3 objects, 2 references)

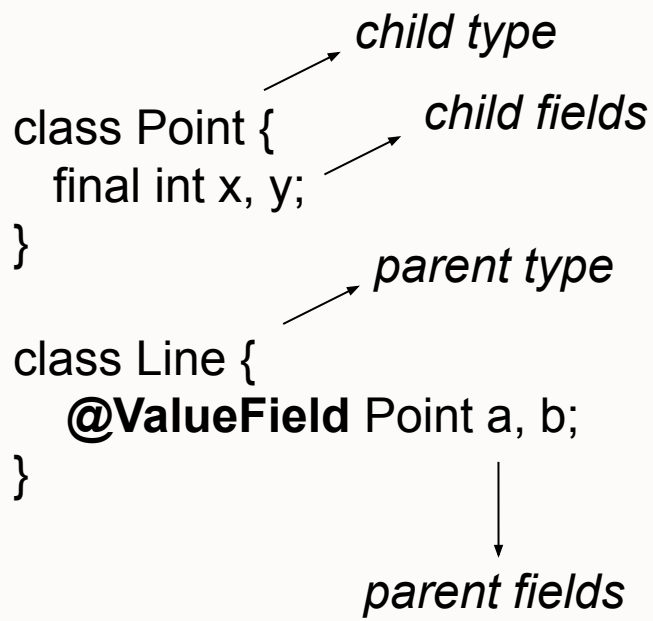
Data *objectification*

Object Inlining

Value Fields

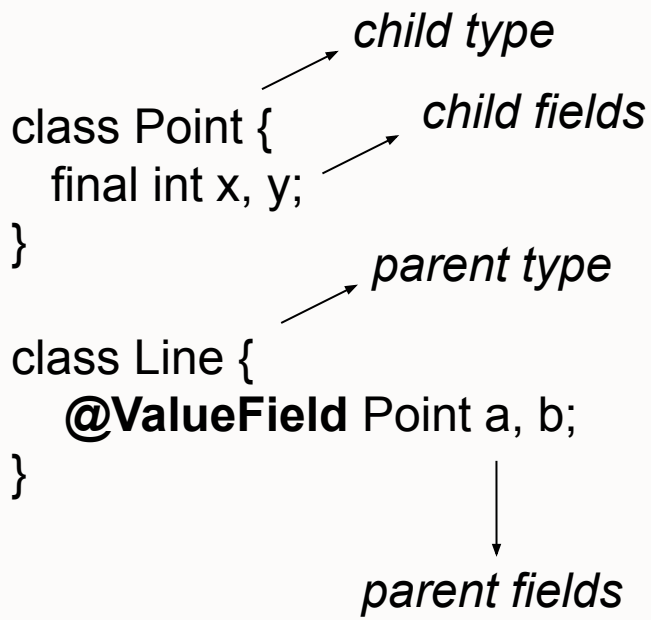
- **Simple abstraction**: fields with value semantics
  - opportunity for optimization (hint for the compiler)
    - fields have do not require identity
    - field accesses have do not require atomic reference access
- **Closed-world Assumption**
  - GraalVM Native Image
  - all types are know at compile time
- **Easy to use**
  - Field annotations: @ValueField, @ValueGraph
  - JSON configuration file
- **Target**
  - Java applications
  - allow framework/library developers to efficient data structures
  - end-users will automatically benefit

Inlined Field Memory Layout

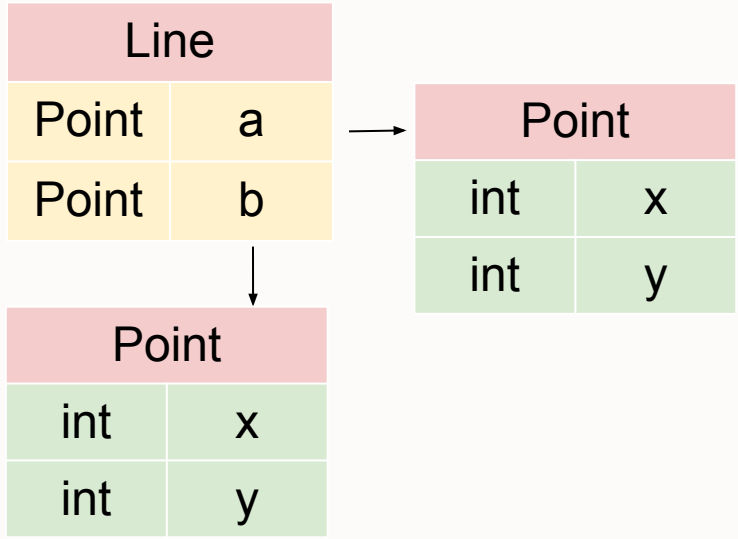


Java source

Inlined Field Memory Layout

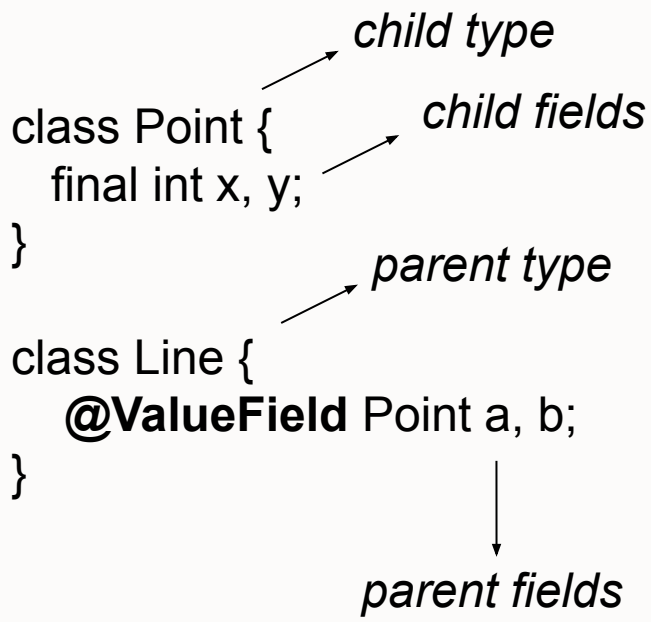


Java source

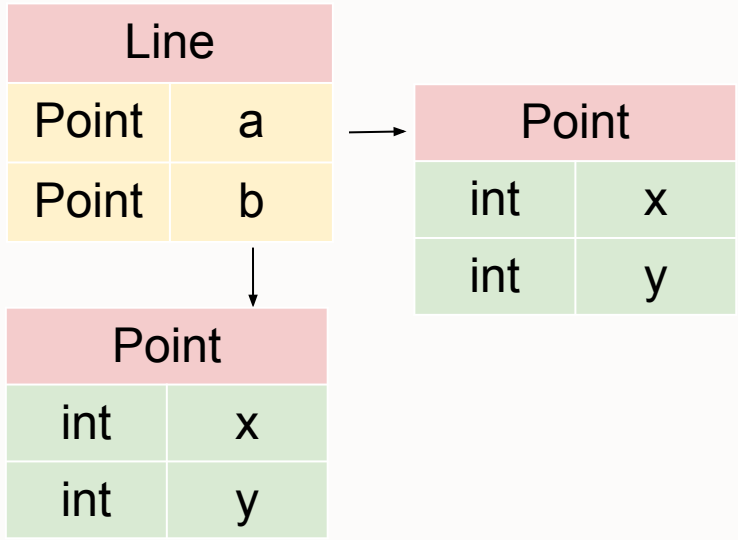


Original Object Layout

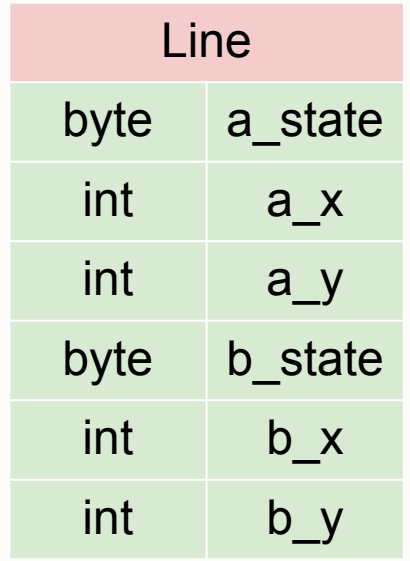
Inlined Field Memory Layout



Java source



Original Object Layout



Inlined Object Layout

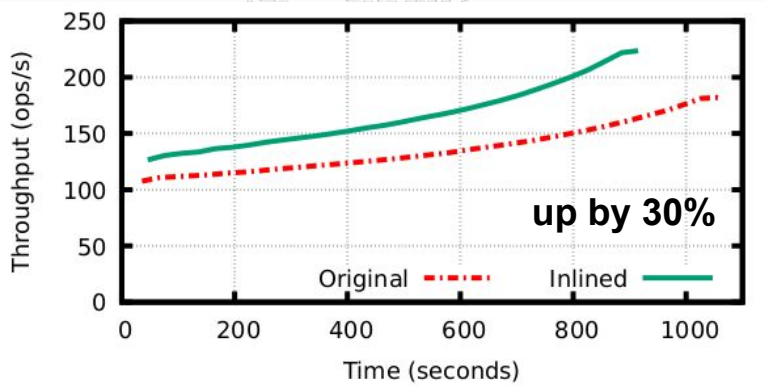
Restrictions of Object Inlining

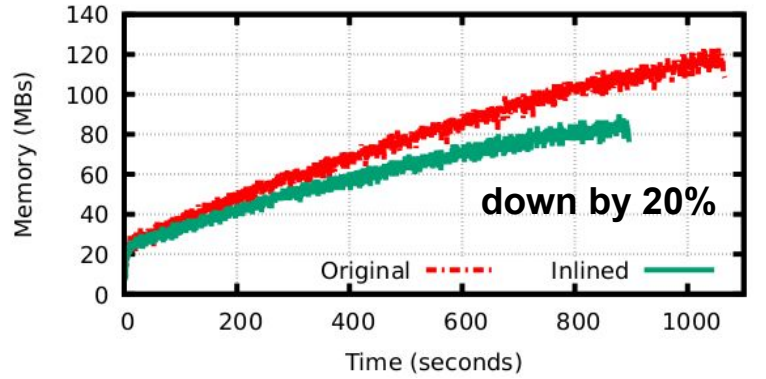
- Inlining is restricted to fields of **immutable** types
  - Modifications are not guaranteed to be propagated after inlining
  - If inlined objects are **not modified** after inlined, it is **safe** to inline **mutable** types
    - @ValueField(allowMutable=true), @ValueGraph(allowMutable=true)
- No **polymorphic** types
  - We need to know the type layout at image build time
- No fields of **primitive** types, **array** type, **volatile**, **static**

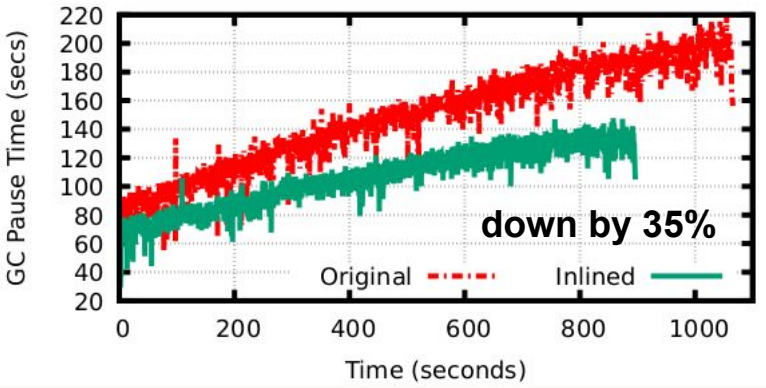
Spring Boot Online Website

- PetClinic workload:
  - Apache Jmeter produces requests
  - Microservice (Spring Boot / Micronaut) handles requests
  - SQL DB persists information
  - Realistic dataset (names of pets, real addresses, etc)
  - Workload consists of read and write requests
- Values are cached in memory (locally or remotely)
- Object inlining configuration file:

```
{
  "value_fields" : {
    "petclinic.model.BaseEntity" : ["id"],
    "petclinic.model.Person" : ["firstName", "lastName"],
    "petclinic.model.NamedEntity" : ["name"],
    "petclinic.owner.Owner" : ["address", "city",  "phone"],
    "petclinic.owner.Pet" : ["birthDate", "type"],
    "petclinic.visit.Visit" : ["date", "desc", "petId"]
  }
}
```







Data Processing in Spark

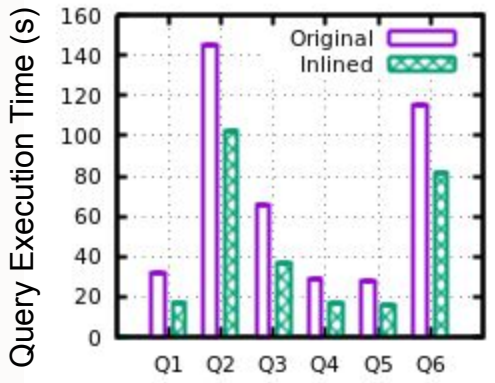
- Data analytics workload:
  - Large JSON dataset (IMDB movie collection)
  - 6 different queries are executed over the dataset
- Each JSON entry is loaded into a Spark RDD
  - Value Inlining can be used to inline fields inside simple POJO types

Query	Description
Q1	Number of movies released in a year by genre.
Q2	Movies ordered by movie rating.
Q3	Average age of a movie's actors.
Q4	Actors ordered by number of roles.
Q5	Year with more average rating vores.
Q6	Actors ordered by the number of roles in highly rated movies.

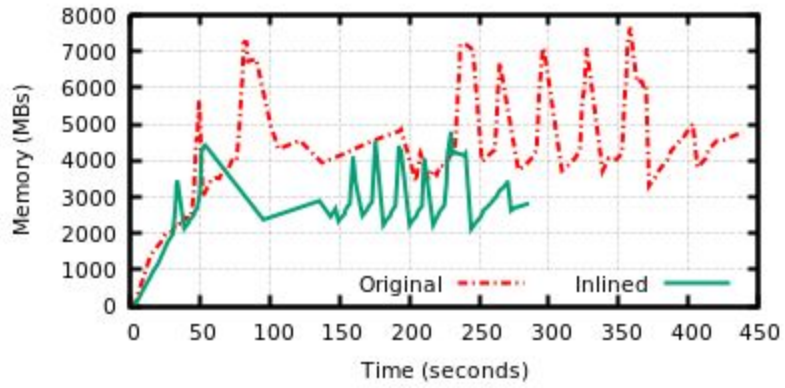
```
1: class Movie {
2:   Actor[] actors;
3:   @ValueField String genre;
4:   @ValueField String name;
5:   @ValueField Date release;
6:   int votes;
7:   float rating;
8: }
```

```
1: class Actor {
2:   @ValueField Date birth;
3:   @ValueField Date death;
4:   @ValueField String name;
5: }
```

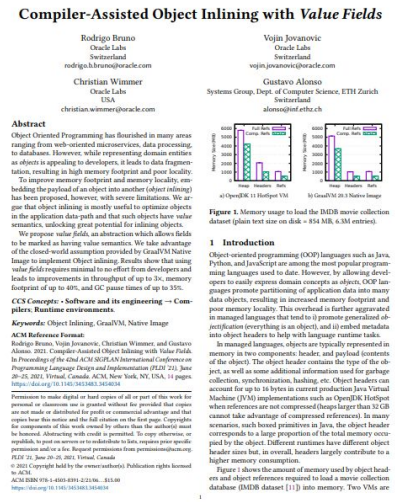
down by 35%



down by 41%



More details available in the paper:



Compiler-Assisted Object Inlining with *Value Fields*

Rodrigo Bruno
Oracle Labs
Switzerland
rodrigo.b.bruno@oracle.com

Vojin Jovanovic
Oracle Labs
Switzerland
vojin.jovanovic@oracle.com

Christian Wimmer
Oracle Labs
USA
christian.wimmer@oracle.com

Gustavo Alonso
Systems Group, Dept. of Computer Science, ETH Zurich
Switzerland
alonso@inf.ethz.ch

Abstract

Object Oriented Programming has flourished in many areas, ranging from web-oriented microservices, data processing, to databases. However, while representing domain entities as objects is appealing to developers, it leads to data fragmentation, resulting in high memory footprint and poor locality.

To improve memory footprint and memory locality, embedding the payload of an object into another (*object inlining*) has been proposed, however, with severe limitations. We argue that object inlining is mostly useful to optimize objects in the application data-path and that such objects have *value* semantics, unlocking great potential for inlining objects.

We propose *value fields*, an abstraction which allows fields to be marked as having value semantics. We take advantage of the closed-world assumption provided by GraalVM Native Image to implement Object inlining. Results show that using *value fields* requires minimal to no effort from developers and leads to improvements in throughput of up to 3×, memory footprint of up to 40%, and GC pause times of up to 35%.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Runtime environments**.

Keywords: Object Inlining, GraalVM, Native Image

ACM Reference Format:

Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. 2021. Compiler-Assisted Object Inlining with *Value Fields*. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453483.3454034

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454034

a) OpenJDK 11 HotSpot VM b) GraalVM 20.3 Native Image

Figure 1. Memory usage to load the IMDB movie collection dataset (plain text size on disk = 854 MB, 6.3M entries).

1 Introduction

Object-oriented programming (OOP) languages such as Java, Python, and JavaScript are among the most popular programming languages used to date. However, by allowing developers to easily express domain concepts as objects, OOP languages promote partitioning of application data into many data objects, resulting in increased memory footprint and poor memory locality. This overhead is further aggravated in managed languages that tend to i) promote generalized *objectification* (everything is an object), and ii) embed metadata into object headers to help with language runtime tasks.

In managed languages, objects are typically represented in memory in two components: header, and payload (contents of the object). The object header contains the type of the object, as well as some additional information used for garbage collection, synchronization, hashing, etc. Object headers can account for up to 16 bytes in current production Java Virtual Machine (JVM) implementations such as OpenJDK HotSpot when references are not compressed (heaps larger than 32 GB cannot take advantage of compressed references). In many scenarios, such boxed primitives in Java, the object header corresponds to a large proportion of the total memory occupied by the object. Different runtimes have different object header sizes but, in overall, headers largely contribute to a higher memory consumption.

Figure 1 shows the amount of memory used by object headers and object references required to load a movie collection database (IMDB dataset [11]) into memory. Two VMs are

Rodrigo Bruno

rodrigo.bruno@tecnico.ulisboa.pt

https://rodrigo-bruno.github.io/

Thank You

ORACLE